technische universität
dortmund

LS 2 /
Informatik

Datenstrukturen, Algorithmen und Programmierung 2 (DAP2)

Big Data

Was ist Big Data?

Der Begriff Big Data bezeichnet Daten, die so groß sind, dass die Größe der Daten zum algorithmischen Problem wird

Volume

Die zu verarbeitenden Datenmengen sind riesig

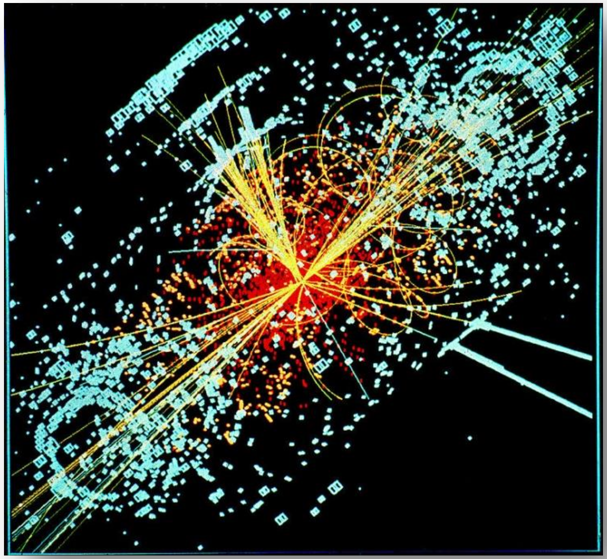
Variety

Die Daten haben keine feste Struktur

Velocity

Hohe Datenraten und/oder Datenströme

Big Data

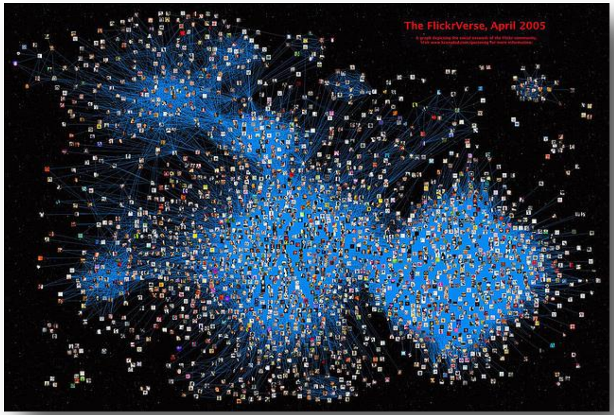


Large Hadron Collider

- Teilchen werden mit annähernd Licht-geschwindigkeit aufeinander geschossen
- Die Bahnen der beim Zerfall entstehenden Teilchen können beobachtet werden
- PetaBytes von Daten (1,000,000 GigaByte)

Lucas Taylor - http://cdsweb.cern.ch/record/628469

Big Data



Soziale Netzwerke

- Können wir anhand des Facebookgraphen eines Landes erkennen, ob das Land eine Demokratie oder ein totalitärer Staat ist?
- GigaByte bis TeraByte (nur Graph)
- PetaByte zusätzlicher Informationen

Big Data

Festplatten

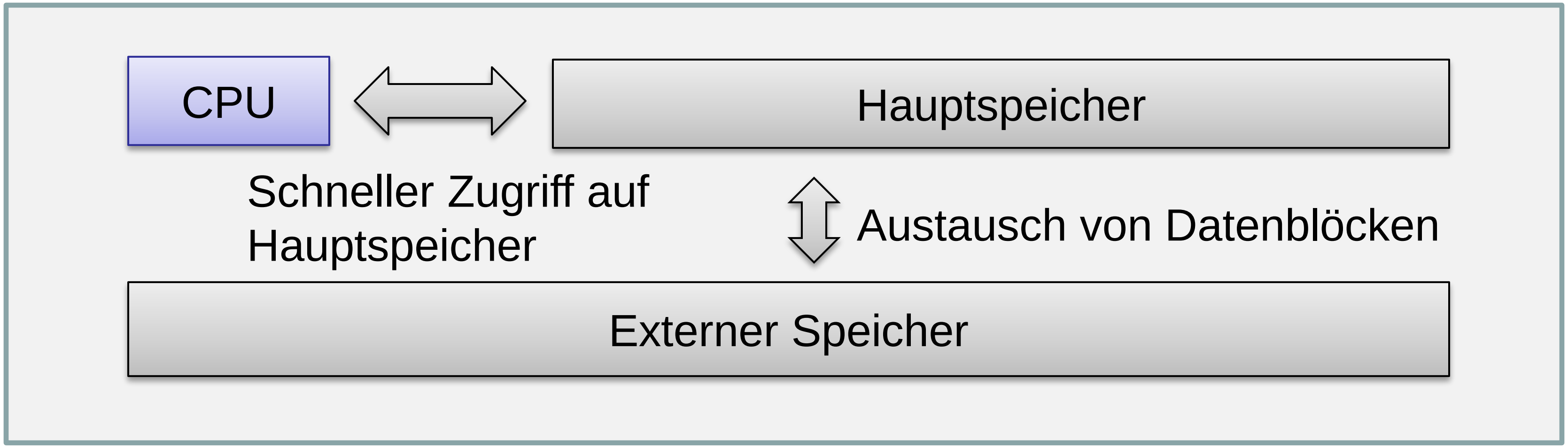
- Daten werden auf rotierender Scheibe durch Magnetisierung gespeichert
- Freier Zugriff benötigt Positionierung eines Schreib-/Lesekopfs
- Dies ist relativ langsam, da mechanisch
- Das Lesen von Daten (nach der Positionierung) ist relativ schnell

Typische Funktionsweise

Schreib- und Lesezugriffe werden daher typischerweise in Datenblöcke (Seiten) von einige KByte organisiert

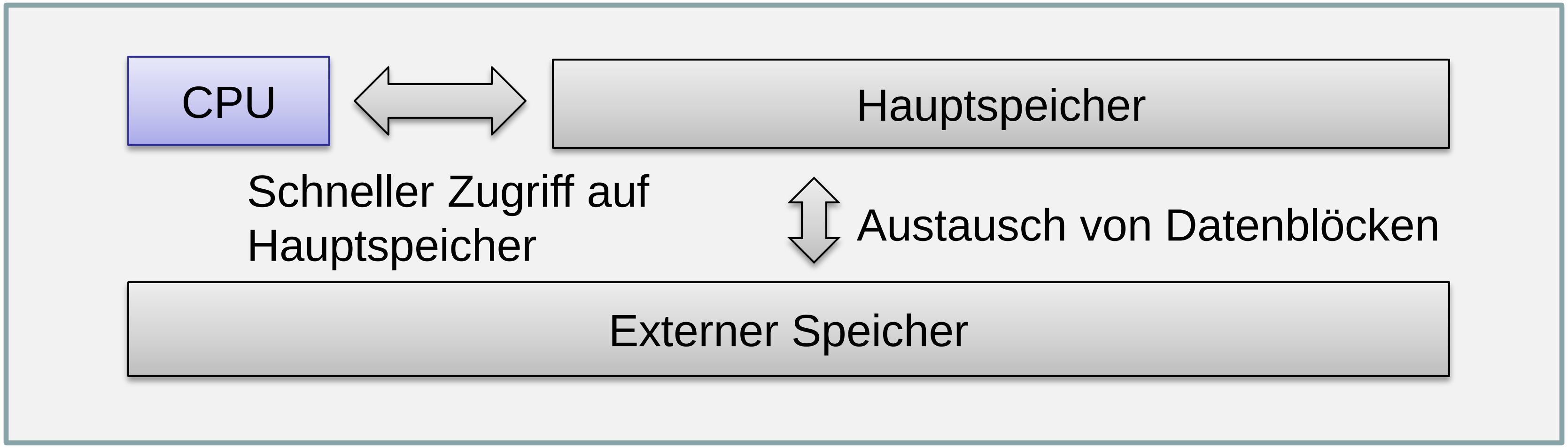
Big Data

Externspeichermodell



Big Data

Externspeichermodell



Zwei Flaschenhälse

- Laufzeit CPU
- Anzahl Zugriffe auf externen Speicher

Big Data

Algorithmenanalyse im Externspeichermodell

- Eingabe ist auf Festplatte
- Disk-Read(x) liest Datum x von Festplatte
- Disk-Write(x) schreibt Datum x auf Festplatte
- Ist x bereits im Speicher, so zählen wir Disk-Read(x) nicht
- (Nur begrenzt viel schneller Speicher)

Qualitätsmaße

- Laufzeit CPU (wie immer)
- Anzahl Disk-Read und Disk-Write Operationen (neu)

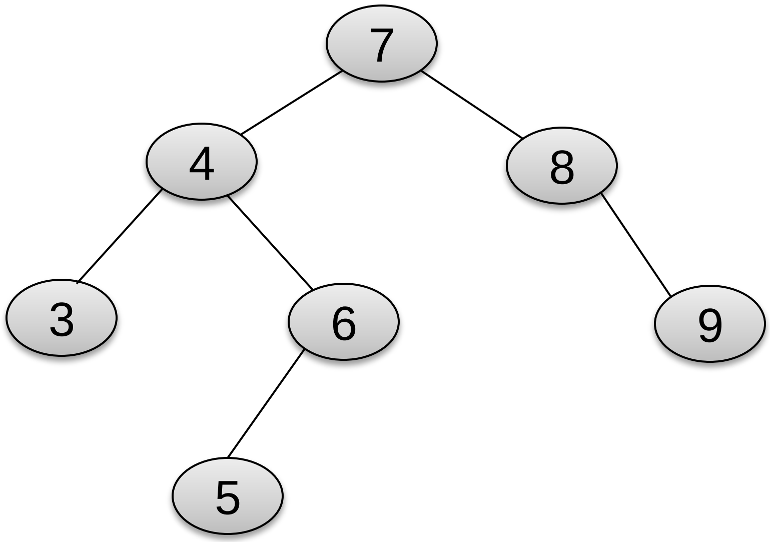
Big Data

B-Bäume

Ziel: Effiziente Suchbaumstruktur im Externspeichermodell

Nachteile von Binärbäumen

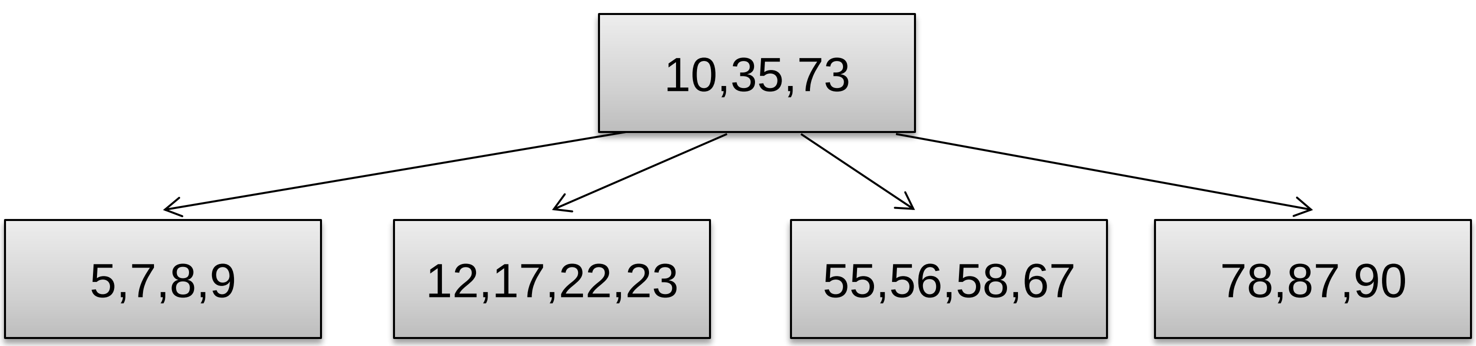
- Bei der Suche in Binärbäumen wird u.U. bei jedem neuen Knoten ein Disk-Read durchgeführt
- Es wird aber nur ein einziger Wert (der Schlüssel) benötigt



Big Data

B-Bäume – Grundidee

- Neuer Suchbaumstruktur mit „größeren Knoten“ und höherem Verzweigungsgrad
- Jeder Knoten enthält aufsteigend sortierte Folge von Schlüsseln
- k Schlüssel an einem Knoten partitionieren das Schlüsseluniversum in $k + 1$ Intervalle
- für jedes solche Intervall gibt es einen Unterbaum, der alle Knoten des Intervalls enthält



Big Data

B-Bäume – Struktur eines Knotens x

- Anzahl gespeicherter Schlüssel $n[x]$
- Die gespeicherten Schlüssel $\mathrm{key}_1[x] \leq \mathrm{key}_2[x] \leq \cdots \leq \mathrm{key}_{n[x]}[x]$ sind aufsteigend sortiert
- Jeder Knoten hat $n[x] + 1$ Zeiger $c_1[x], c_2[x], \dots, c_{n[x]+1}[x]$ auf seine Kinder (**nil**, wenn nicht existent)
- Die Schlüssel und die Zeiger sind jeweils in einem Feld mit $2t - 1$ bzw. $2t$ Einträgen gespeichert (für Parameter t)

Big Data

B-Bäume – Struktur des Baums

- Parameter $t \approx$ Größe eines Datenblocks
- Jeder Knoten hat höchstens $2t - 1$ Schlüssel
- Die Wurzel eines nichtleeren B-Baums hat mindestens 1 Schlüssel
- Jeder andere Knoten hat mindestens $t - 1$ Schlüssel
- Der Baum ist balanciert (jeder Blattknoten hat dieselbe Höhe)

Big Data

Lemma 38

Für einen B-Baum mit Parameter t, $n \geq 1$ Schlüsseln und Höhe h gilt $h \leq \log_t\big((n+1)/2\big)$.

Beweis

- Die Wurzel hat mindestens einen Schlüssel und jeder andere Knoten mindestens $t-1$ Schlüssel

Big Data

Lemma 38

Für einen B-Baum mit Parameter t, $n \geq 1$ Schlüsseln und Höhe h gilt $h \leq \log_t\big((n+1)/2\big)$.

Beweis

- Die Wurzel hat mindestens einen Schlüssel und jeder andere Knoten mindestens $t-1$ Schlüssel
- Es gibt also mind. 2 Knoten der Höhe 1 und mind. $2t$ Knoten der Höhe 2, $2t^2$ Knoten der Höhe 3, usw.

Big Data

Lemma 38

Für einen B-Baum mit Parameter t, $n \geq 1$ Schlüsseln und Höhe h gilt $h \leq \log_t\big((n+1)/2\big)$.

Beweis

- Die Wurzel hat mindestens einen Schlüssel und jeder andere Knoten mindestens $t-1$ Schlüssel
- Es gibt also mind. 2 Knoten der Höhe 1 und mind. $2t$ Knoten der Höhe 2, $2t^2$ Knoten der Höhe 3, usw.
- $n \geq 1 + (t-1) \cdot \sum_{i=1}^{h} 2t^{i-1} = 1 + 2 \cdot (t-1) \cdot \left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1$

Big Data

Lemma 38

Für einen B-Baum mit Parameter t, $n \geq 1$ Schlüsseln und Höhe h gilt
$h \leq \log_t\big((n+1)/2\big)$.

Beweis

- Die Wurzel hat mindestens einen Schlüssel und jeder andere Knoten mindestens $t-1$ Schlüssel
- Es gibt also mind. 2 Knoten der Höhe 1 und mind. $2t$ Knoten der Höhe 2, $2t^2$ Knoten der Höhe 3, usw.
- $n \geq 1 + (t-1) \cdot \sum_{i=1}^{h} 2t^{i-1} = 1 + 2 \cdot (t-1) \cdot \left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1$
- Es folgt $t^h \leq (n+1)/2$ und somit $h \leq \log_t((n+1)/2)$.

Big Data

Lemma 38

Für einen B-Baum mit Parameter t, $n \geq 1$ Schlüsseln und Höhe h gilt $h \leq \log_t\big((n+1)/2\big)$.

Beweis

- Die Wurzel hat mindestens einen Schlüssel und jeder andere Knoten mindestens $t-1$ Schlüssel
- Es gibt also mind. 2 Knoten der Höhe 1 und mind. $2t$ Knoten der Höhe 2, $2t^2$ Knoten der Höhe 3, usw.
- $n \geq 1 + (t-1) \cdot \sum_{i=1}^{h} 2t^{i-1} = 1 + 2 \cdot (t-1) \cdot \left(\frac{t^h-1}{t-1}\right) = 2t^h - 1$
- Es folgt $t^h \leq (n+1)/2$ und somit $h \leq \log_t((n+1)/2)$.

Big Data

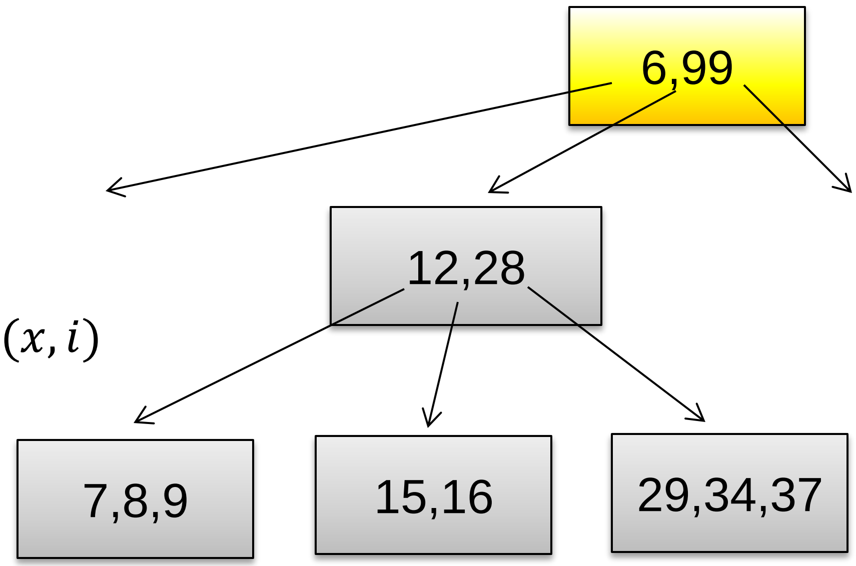
B-BaumSuche(x, k)

$i \leftarrow 1$
while $i \leq n[x]$ and $k > key_i[x]$ **do**
    $i \leftarrow i + 1$
if $i \leq n[x]$ and $k = key_i[x]$ **then return** (x, i)
if x is a leaf **then return nil**
else
    Disk-Read$(c_i[x])$
    return B-BaumSuche$(c_i[x], k)$

<u>Grundidee:</u>
Wie Suche in Binärbäumen, aber statt einer 2-Wege Entscheidung führt man eine Mehrwegeentscheidung durch

Big Data

Suche(root,15)



B-BaumSuche(x,k)

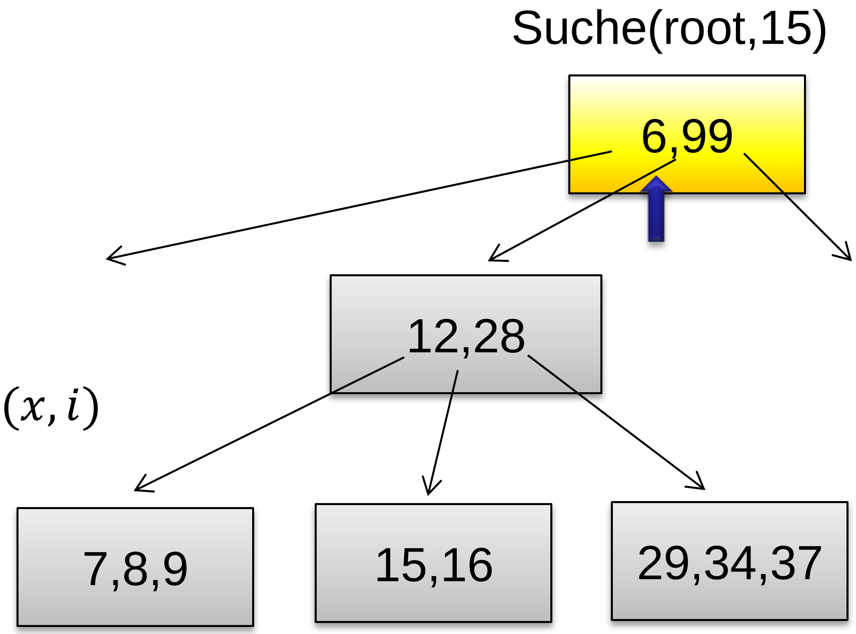
```
i ← 1
while i ≤ n[x] and k > key_i[x] do
    i ← i + 1
if i ≤ n[x] and k = key_i[x] then return (x, i)
if x is a leaf then return nil
else
    Disk-Read(c_i[x])
    return B-BaumSuche(c_i[x], k)
```

<u>Grundidee:</u>
Wie Suche in Binärbäumen, aber statt einer 2-Wege Entscheidung führt man eine Mehrwegeentscheidung durch

Big Data

B-BaumSuche(x, k)

```
i ← 1
while i ≤ n[x] and k > key_i[x] do
    i ← i + 1
if i ≤ n[x] and k = key_i[x] then return (x, i)
if x is a leaf then return nil
else
    Disk-Read(c_i[x])
    return B-BaumSuche(c_i[x], k)
```



<u>Grundidee:</u>
Wie Suche in Binärbäumen, aber statt einer 2-Wege Entscheidung führt man eine Mehrwegeentscheidung durch

Big Data

B-BaumSuche(x, k)

$i \leftarrow 1$

while $i \leq n[x]$ and $k > key_i[x]$ **do**

  $i \leftarrow i + 1$

if $i \leq n[x]$ and $k = key_i[x]$ **then return** (x, i)

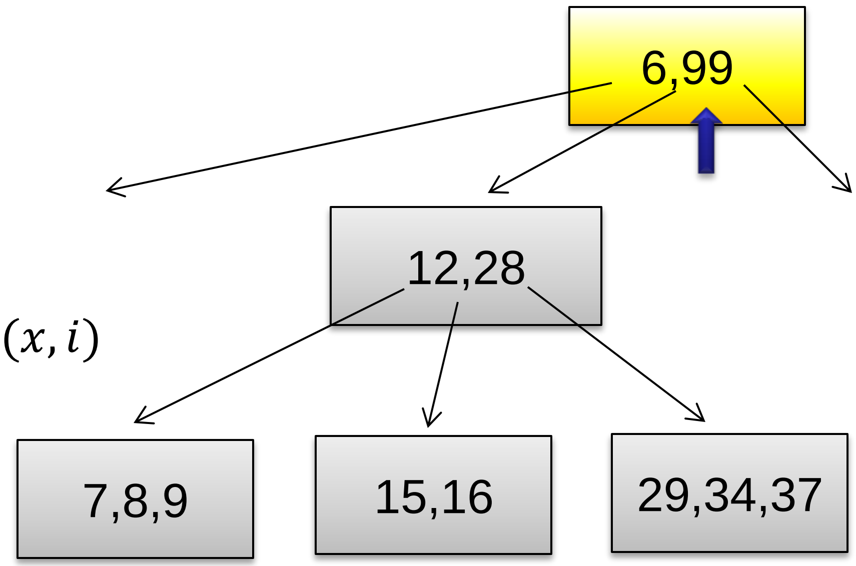
if x is a leaf **then return nil**

else

  Disk-Read$(c_i[x])$

  return B-BaumSuche$(c_i[x], k)$

Suche(root,15)



<u>Grundidee:</u>
Wie Suche in Binärbäumen, aber statt einer 2-Wege Entscheidung führt man eine Mehrwegeentscheidung durch

Big Data

B-BaumSuche(x, k)

$i \leftarrow 1$

while $i \leq n[x]$ and $k > key_i[x]$ **do**

   $i \leftarrow i + 1$

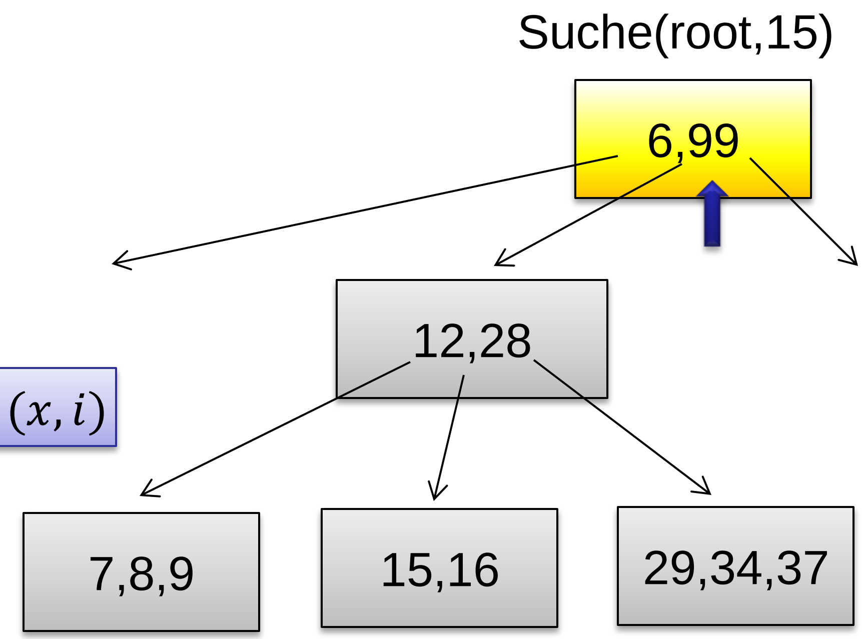
if $i \leq n[x]$ and $k = key_i[x]$ **then return** (x, i)

if x is a leaf **then return nil**

else

   Disk-Read$(c_i[x])$

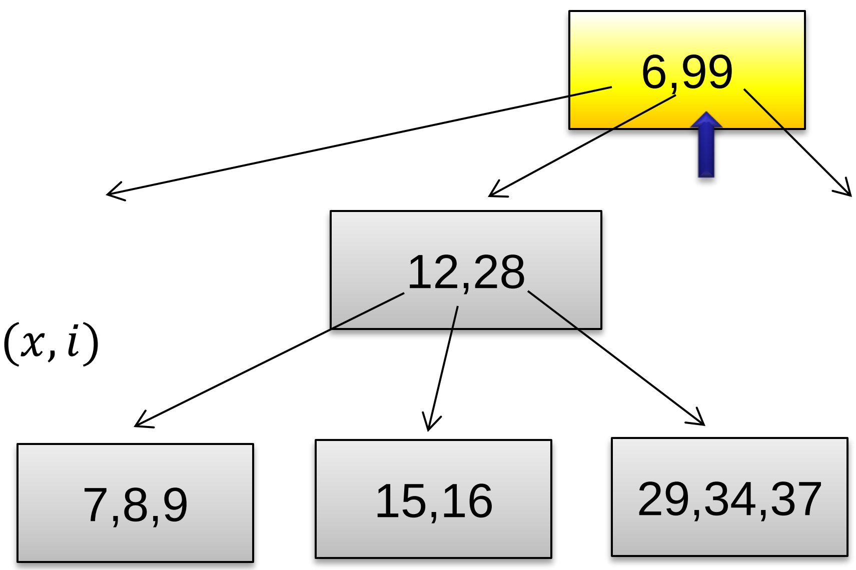
   return B-BaumSuche$(c_i[x], k)$



<u>Grundidee:</u>
Wie Suche in Binärbäumen, aber statt einer 2-Wege Entscheidung führt man eine Mehrwegeentscheidung durch

Big Data

Suche(root,15)



B-BaumSuche(x, k)

$i \leftarrow 1$

while $i \leq n[x]$ and $k > key_i[x]$ **do**

  $i \leftarrow i + 1$

if $i \leq n[x]$ and $k = key_i[x]$ **then return** (x, i)

if x is a leaf **then return nil**

else

  Disk-Read$(c_i[x])$

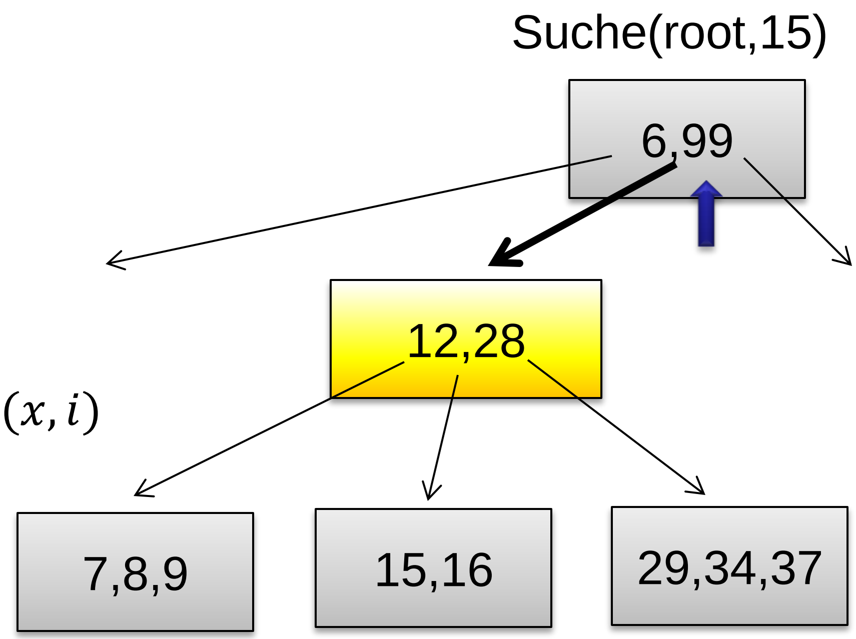
  return B-BaumSuche$(c_i[x], k)$

<u>Grundidee:</u>
Wie Suche in Binärbäumen, aber statt einer 2-Wege Entscheidung führt man eine Mehrwegeentscheidung durch

Big Data

B-BaumSuche(x, k)

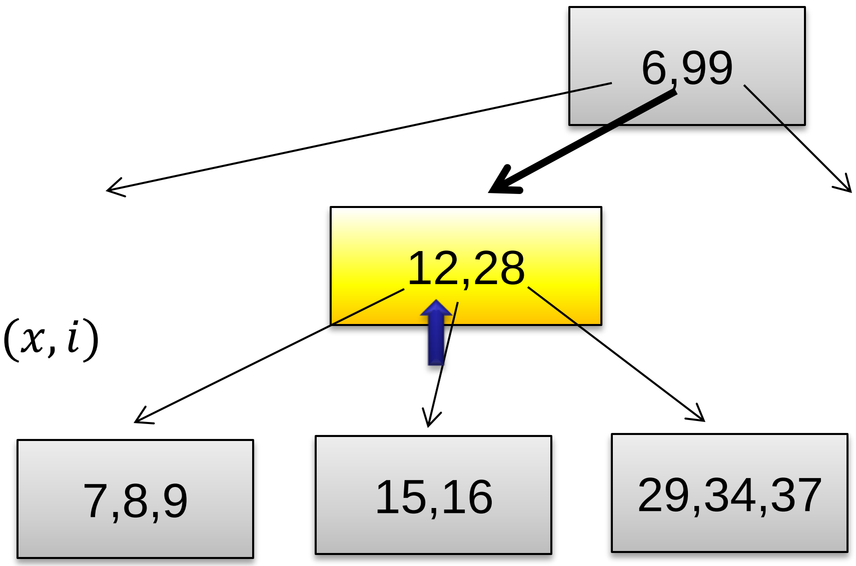
```
i ← 1
while i ≤ n[x] and k > key_i[x] do
    i ← i + 1
if i ≤ n[x] and k = key_i[x] then return (x, i)
if x is a leaf then return nil
else
    Disk-Read(c_i[x])
    return B-BaumSuche(c_i[x], k)
```



Grundidee:
Wie Suche in Binärbäumen, aber statt einer 2-Wege Entscheidung führt man eine Mehrwegeentscheidung durch

Big Data

Suche(root,15)



B-BaumSuche(x, k)

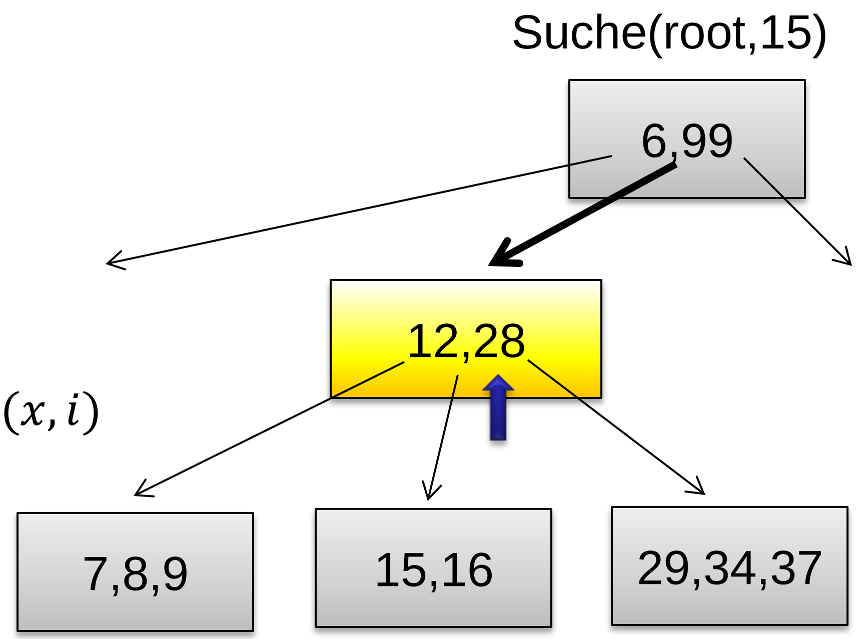
```
i ← 1
while i ≤ n[x] and k > key_i[x] do
    i ← i + 1
if i ≤ n[x] and k = key_i[x] then return (x, i)
if x is a leaf then return nil
else
    Disk-Read(c_i[x])
    return B-BaumSuche(c_i[x], k)
```

<u>Grundidee:</u>
Wie Suche in Binärbäumen, aber statt einer 2-Wege Entscheidung führt man eine Mehrwegeentscheidung durch

Big Data

B-BaumSuche(x, k)

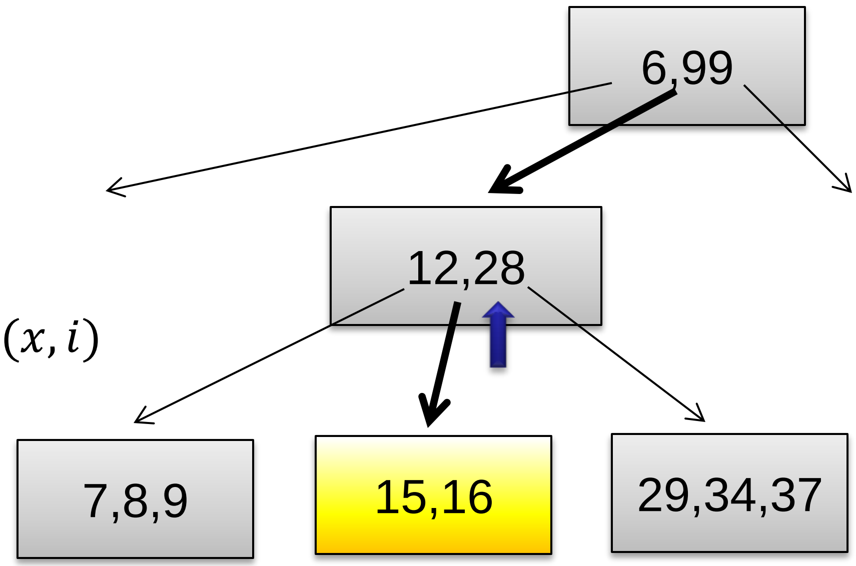
$i \leftarrow 1$
while $i \leq n[x]$ and $k > key_i[x]$ **do**
    $i \leftarrow i + 1$
if $i \leq n[x]$ and $k = key_i[x]$ **then return** (x, i)
if x is a leaf **then return nil**
else
    Disk-Read$(c_i[x])$
    return B-BaumSuche$(c_i[x], k)$



<u>Grundidee:</u>
Wie Suche in Binärbäumen, aber statt einer 2-Wege Entscheidung führt man eine Mehrwegeentscheidung durch

Big Data

Suche(root,15)



B-BaumSuche(x, k)

$i \leftarrow 1$
while $i \leq n[x]$ and $k > key_i[x]$ **do**
    $i \leftarrow i + 1$
if $i \leq n[x]$ and $k = key_i[x]$ **then return** (x, i)
if x is a leaf **then return nil**
else
    Disk-Read$(c_i[x])$
    return B-BaumSuche$(c_i[x], k)$

<u>Grundidee:</u>
Wie Suche in Binärbäumen, aber statt einer 2-Wege Entscheidung führt man eine Mehrwegeentscheidung durch

Big Data

B-BaumSuche(x, k)

$i \leftarrow 1$

while $i \leq n[x]$ and $k > key_i[x]$ **do**

    $i \leftarrow i + 1$

if $i \leq n[x]$ and $k = key_i[x]$ **then return** (x, i)

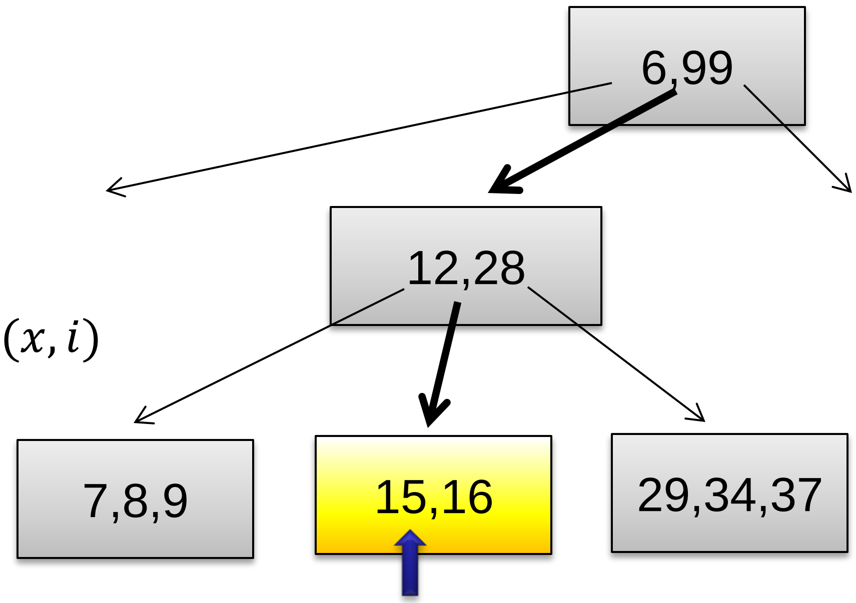
if x is a leaf **then return nil**

else

    Disk-Read$(c_i[x])$

    return B-BaumSuche$(c_i[x], k)$

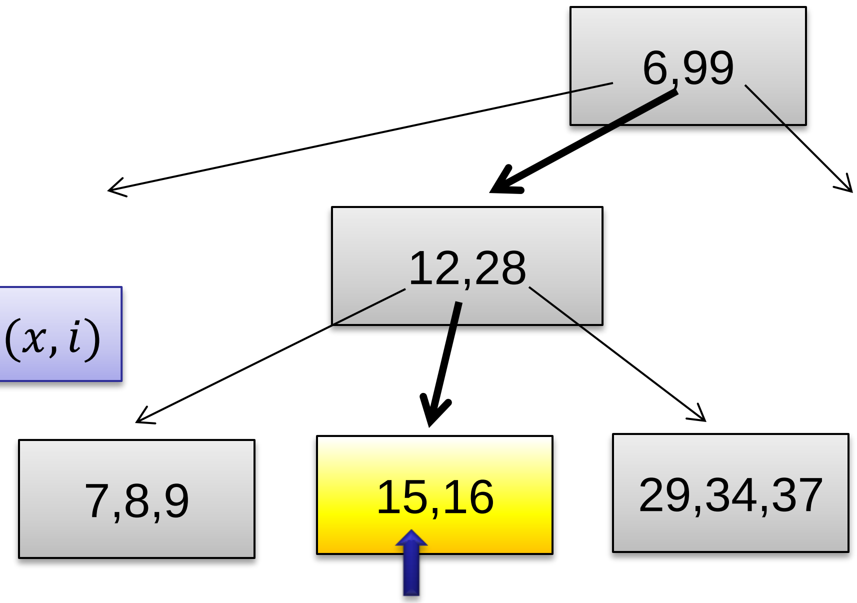
Suche(root,15)



<u>Grundidee:</u>
Wie Suche in Binärbäumen, aber statt einer 2-Wege Entscheidung führt man eine Mehrwegeentscheidung durch

Big Data

Suche(root,15)



B-BaumSuche(x, k)

$i \leftarrow 1$

while $i \leq n[x]$ and $k > key_i[x]$ **do**

  $i \leftarrow i + 1$

if $i \leq n[x]$ and $k = key_i[x]$ **then return** (x, i)

if x is a leaf **then return nil**

else

  Disk-Read$(c_i[x])$

  return B-BaumSuche$(c_i[x], k)$

<u>Grundidee:</u>
Wie Suche in Binärbäumen, aber statt einer 2-Wege Entscheidung führt man eine Mehrwegeentscheidung durch

Big Data

B-BaumSuche(x, k)

$i \leftarrow 1$
while $i \leq n[x]$ and $k > key_i[x]$ **do**
  $i \leftarrow i + 1$
if $i \leq n[x]$ and $k = key_i[x]$ **then return** (x, i)
if x is a leaf **then return nil**
else
  Disk-Read$(c_i[x])$
  return B-BaumSuche$(c_i[x], k)$

Laufzeiten

$\mathbf{O}(th) = \mathbf{O}(t \log_t(n))$ (CPU-)Laufzeit und $\mathbf{O}(\log_t(n))$ Externspeicherzugriffe

Big Data

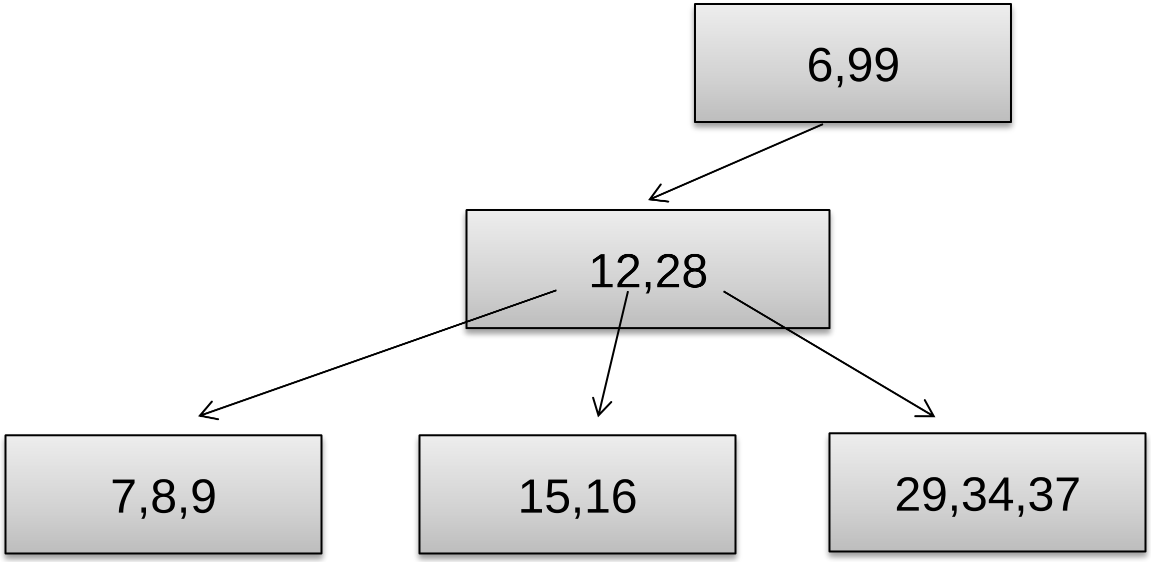
Einfügen in B-Bäumen - Überblick

- Suche nach einzufügendem Schlüssel
- Während der Suche: Wenn ein Knoten auf dem Suchpfad bereits voll ist ($2t-1$ Schlüssel), dann teile diesen Knoten „in der Mitte“ und füge den Medianknoten (den t größten Knoten) in den Vaterknoten ein
- Füge den Knoten in das gefundene Blatt ein

Big Data

Einfügen in B-Bäumen – der einfache Fall ($t = 2$; kein Knoten voll)

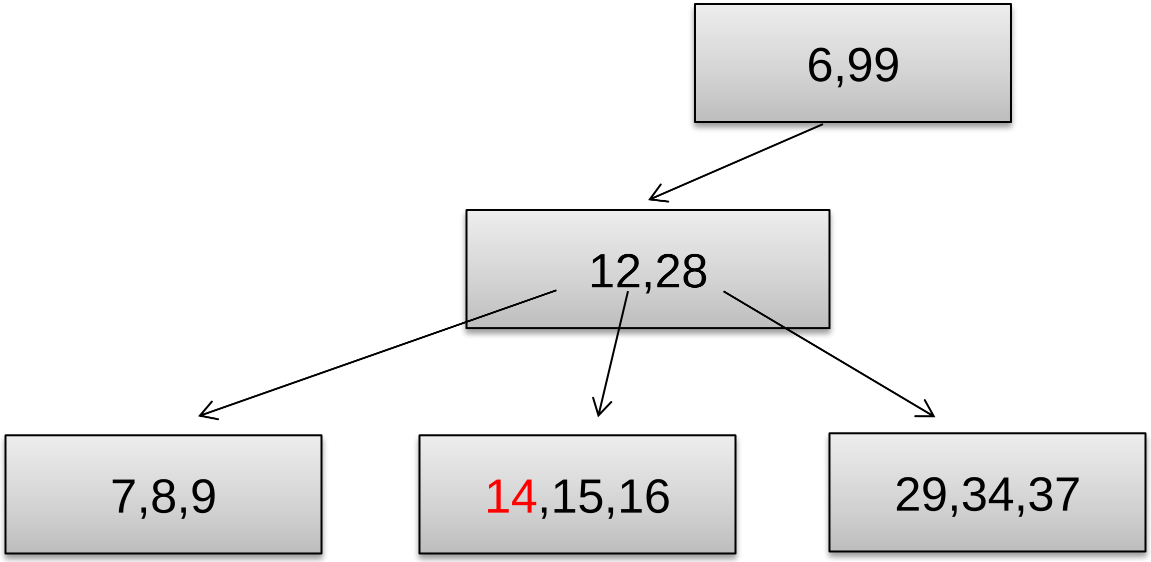
Einfügen von Schlüssel 14



Big Data

Einfügen in B-Bäumen – der einfache Fall ($t = 2$; kein Knoten voll)

Einfügen von Schlüssel 14



Big Data

Aufteilen eines vollen Knotens x

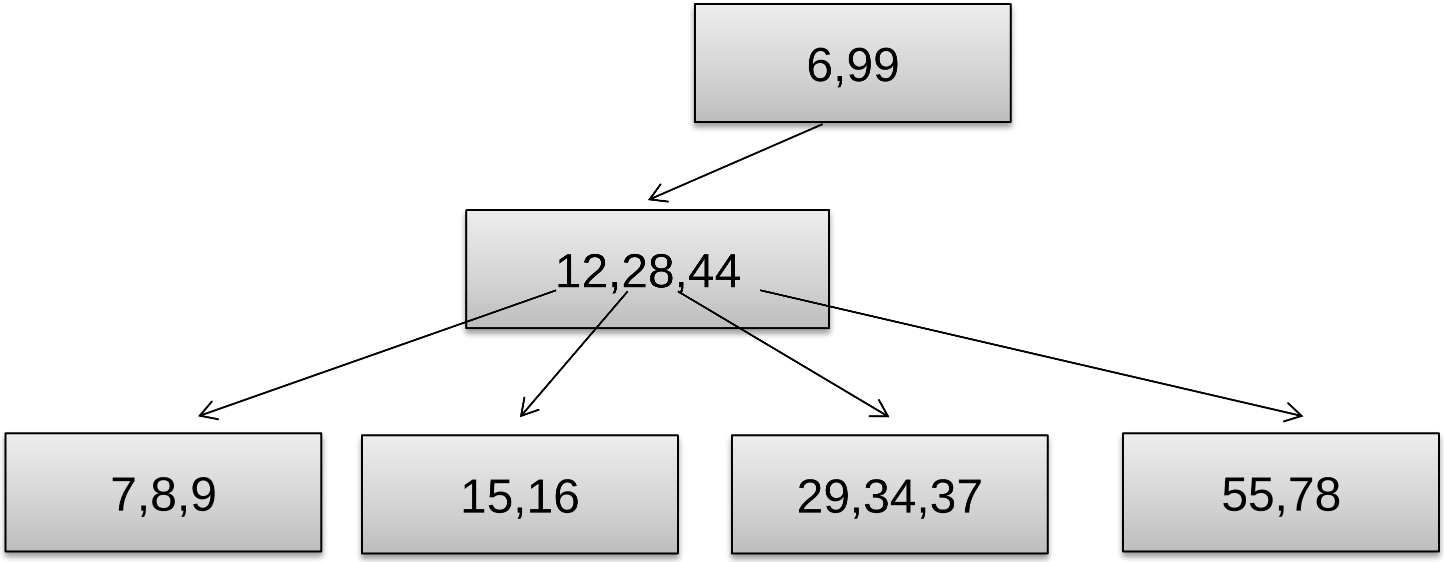
- Teile Schlüsselmenge von x in die $(t-1)$ kleinsten Schlüssel, den Medianschlüssel (den t-kleinsten) und die $(t-1)$ größten Schlüssel auf
- Erzeuge neuen Knoten z und speichere die $(t-1)$ größten Schlüssel von x in z
- Speichere den Medianschlüssel im Vaterknoten
- Lösche die t größten Schlüssel aus x

Aufteilen des Wurzelknotens

Ist der Wurzelknoten voll, so erzeuge einen neuen Wurzelknoten und speichere dort den Medianschlüssel

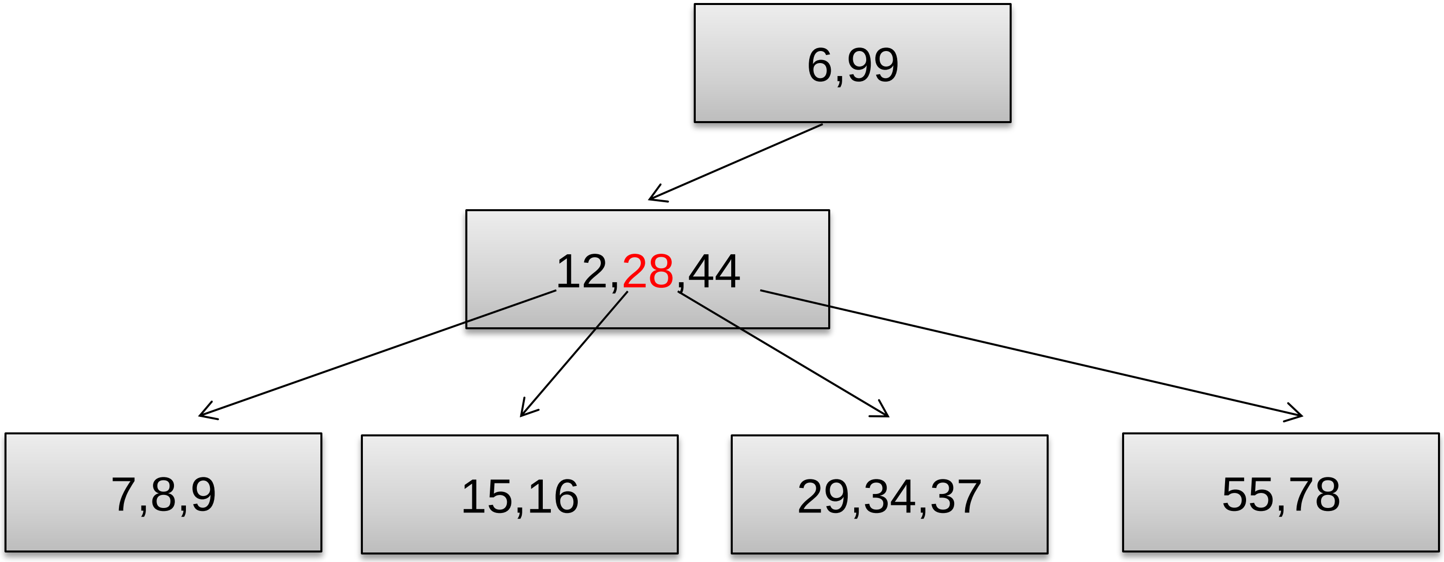
Big Data

Aufteilen eines vollen Knotens ($t = 2$)



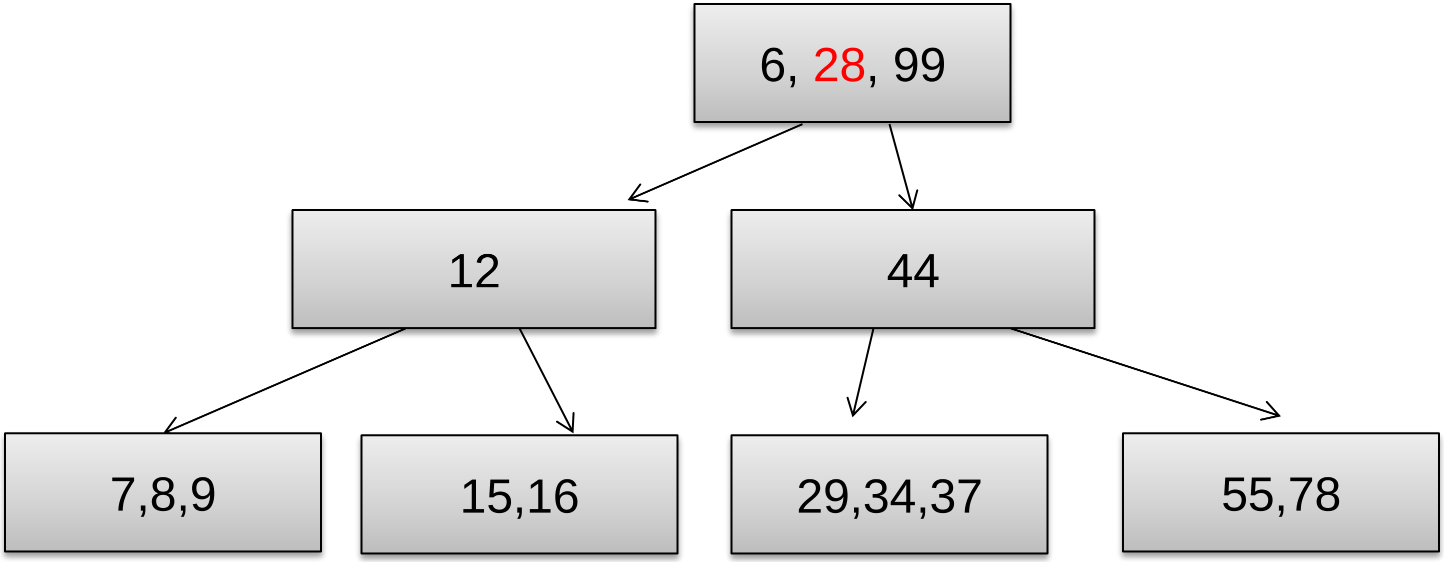
Big Data

Aufteilen eines vollen Knotens ($t = 2$)



Big Data

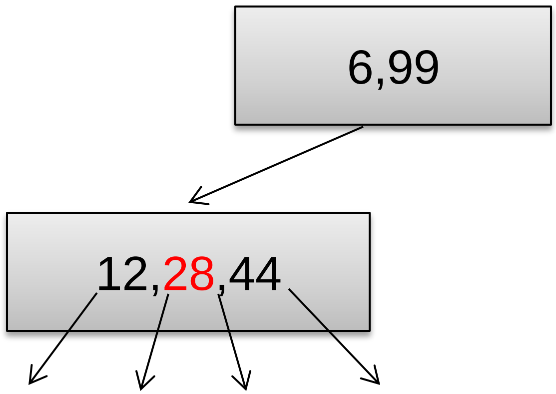
Aufteilen eines vollen Knotens ($t = 2$)



Big Data

Split(x, i, y) ➢ Teilt Knoten y; y ist i-tes Kind von x

1. $z \leftarrow$ Allocate-Node()
2. $n[z] \leftarrow t - 1$
3. Kopiere $\text{key}_{t+1}[y], \ldots, \text{key}_{2t-1}[y]$ nach z
4. Kopiere $c_{t+1}[y], \ldots, c_{2t}[y]$ nach z
5. $n[y] \leftarrow t - 1$
6. **for** $j \leftarrow n[x] + 1$ **downto** $i + 1$ **do** $c_{j+1}[x] \leftarrow c_j[x]$
7. $c_{i+1}[x] \leftarrow z$
8. **for** $j \leftarrow n[x]$ **downto** i **do** $\text{key}_{j+1}[x] \leftarrow \text{key}_j[x]$
9. $\text{key}_i[x] \leftarrow \text{key}_t[y]$
10. $n[x] \leftarrow n[x] + 1$
11. Disk-Write(y); Disk-Write(x); Disk-Write(z)

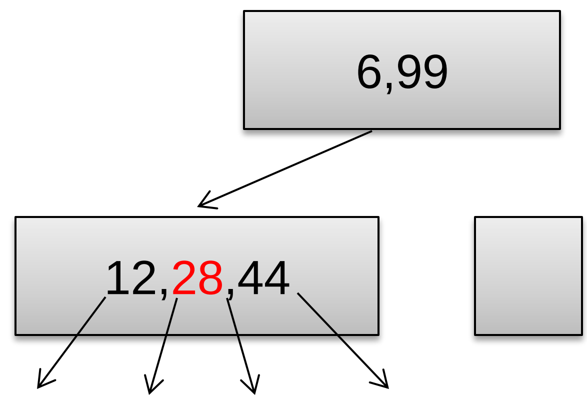


Big Data

Split(x, i, y)

➢ Teilt Knoten y; y ist i-tes Kind von x

1. $\boxed{z \leftarrow \text{Allocate-Node}()}$
2. $n[z] \leftarrow t - 1$
3. Kopiere $\text{key}_{t+1}[y], \dots, \text{key}_{2t-1}[y]$ nach z
4. Kopiere $c_{t+1}[y], \dots, c_{2t}[y]$ nach z
5. $n[y] \leftarrow t - 1$
6. **for** $j \leftarrow n[x] + 1$ **downto** $i + 1$ **do** $c_{j+1}[x] \leftarrow c_j[x]$
7. $c_{i+1}[x] \leftarrow z$
8. **for** $j \leftarrow n[x]$ **downto** i **do** $\text{key}_{j+1}[x] \leftarrow \text{key}_j[x]$
9. $\text{key}_i[x] \leftarrow \text{key}_t[y]$
10. $n[x] \leftarrow n[x] + 1$
11. Disk-Write(y); Disk-Write(x); Disk-Write(z)



Big Data

Split(x, i, y)

➢ Teilt Knoten y; y ist i-tes Kind von x

$z \leftarrow$ Allocate-Node()

$n[z] \leftarrow t - 1$

Kopiere $\mathrm{key}_{t+1}[y], \dots, \mathrm{key}_{2t-1}[y]$ nach z

Kopiere $c_{t+1}[y], \dots, c_{2t}[y]$ nach z

$n[y] \leftarrow t - 1$

for $j \leftarrow n[x] + 1$ **downto** $i + 1$ **do** $c_{j+1}[x] \leftarrow c_j[x]$

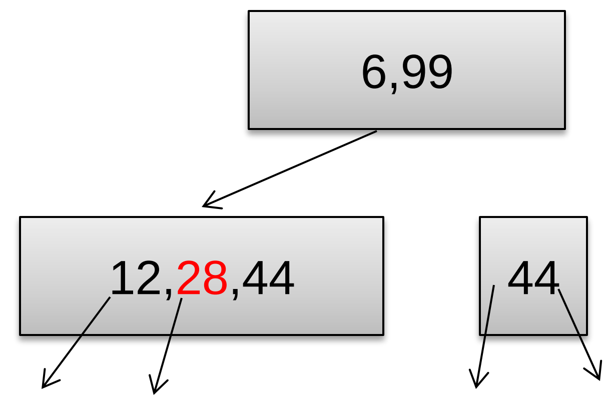
$c_{i+1}[x] \leftarrow z$

for $j \leftarrow n[x]$ **downto** i **do** $\mathrm{key}_{j+1}[x] \leftarrow \mathrm{key}_j[x]$

$\mathrm{key}_i[x] \leftarrow \mathrm{key}_t[y]$

$n[x] \leftarrow n[x] + 1$

Disk-Write(y); Disk-Write(x); Disk-Write(z)

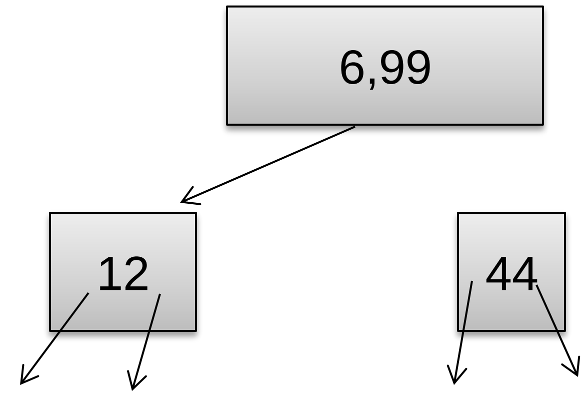


Big Data

Split(x, i, y)

➢ Teilt Knoten y; y ist i-tes Kind von x

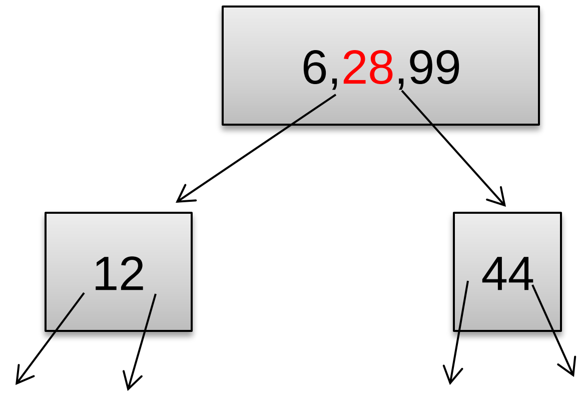
1. $z \leftarrow$ Allocate-Node()
2. $n[z] \leftarrow t - 1$
3. Kopiere $\text{key}_{t+1}[y], \ldots, \text{key}_{2t-1}[y]$ nach z
4. Kopiere $c_{t+1}[y], \ldots, c_{2t}[y]$ nach z
5. $\boxed{n[y] \leftarrow t - 1}$
6. **for** $j \leftarrow n[x] + 1$ **downto** $i + 1$ **do** $c_{j+1}[x] \leftarrow c_j[x]$
7. $c_{i+1}[x] \leftarrow z$
8. **for** $j \leftarrow n[x]$ **downto** i **do** $\text{key}_{j+1}[x] \leftarrow \text{key}_j[x]$
9. $\text{key}_i[x] \leftarrow \text{key}_t[y]$
10. $n[x] \leftarrow n[x] + 1$
11. Disk-Write(y); Disk-Write(x); Disk-Write(z)



Big Data

Split(x, i, y) ➢ Teilt Knoten y; y ist i-tes Kind von x

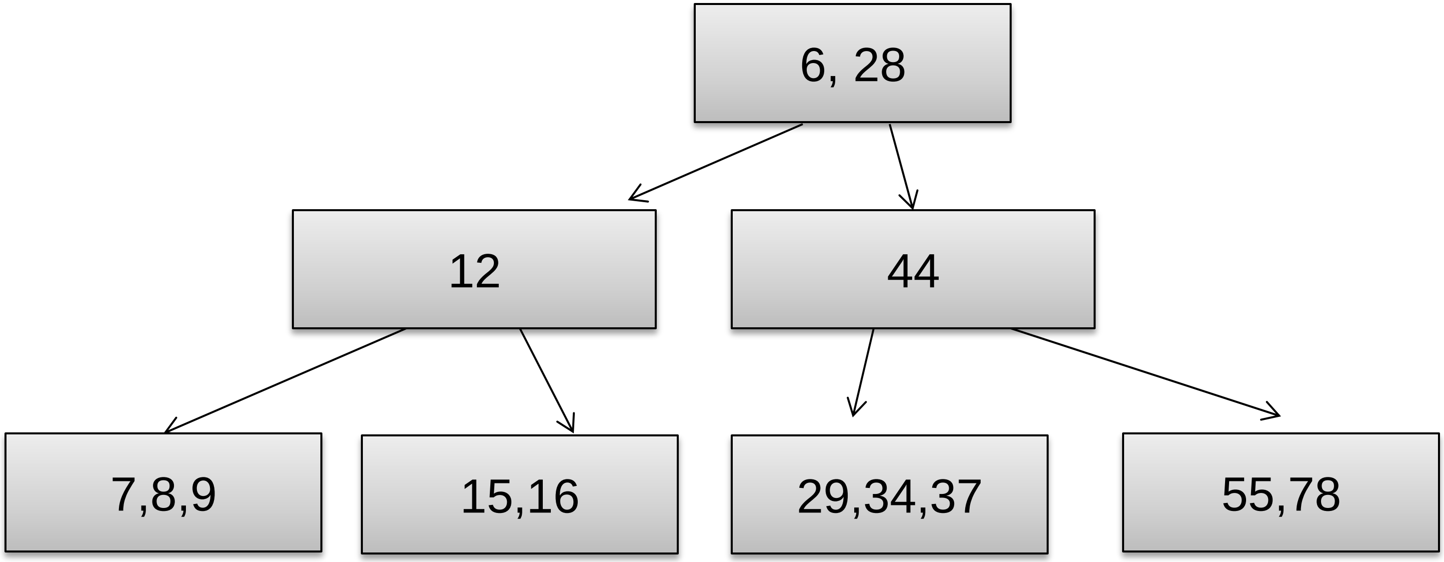
1. $z \leftarrow$ Allocate-Node()
2. $n[z] \leftarrow t - 1$
3. Kopiere $\text{key}_{t+1}[y], \ldots, \text{key}_{2t-1}[y]$ nach z
4. Kopiere $c_{t+1}[y], \ldots, c_{2t}[y]$ nach z
5. $n[y] \leftarrow t - 1$
6. **for** $j \leftarrow n[x] + 1$ **downto** $i + 1$ **do** $c_{j+1}[x] \leftarrow c_j[x]$
7. $c_{i+1}[x] \leftarrow z$
8. **for** $j \leftarrow n[x]$ **downto** i **do** $\text{key}_{j+1}[x] \leftarrow \text{key}_j[x]$
9. $\text{key}_i[x] \leftarrow \text{key}_t[y]$
10. $n[x] \leftarrow n[x] + 1$
11. Disk-Write(y); Disk-Write(x); Disk-Write(z)



Big Data

Einfügen mit Knotenaufteilung

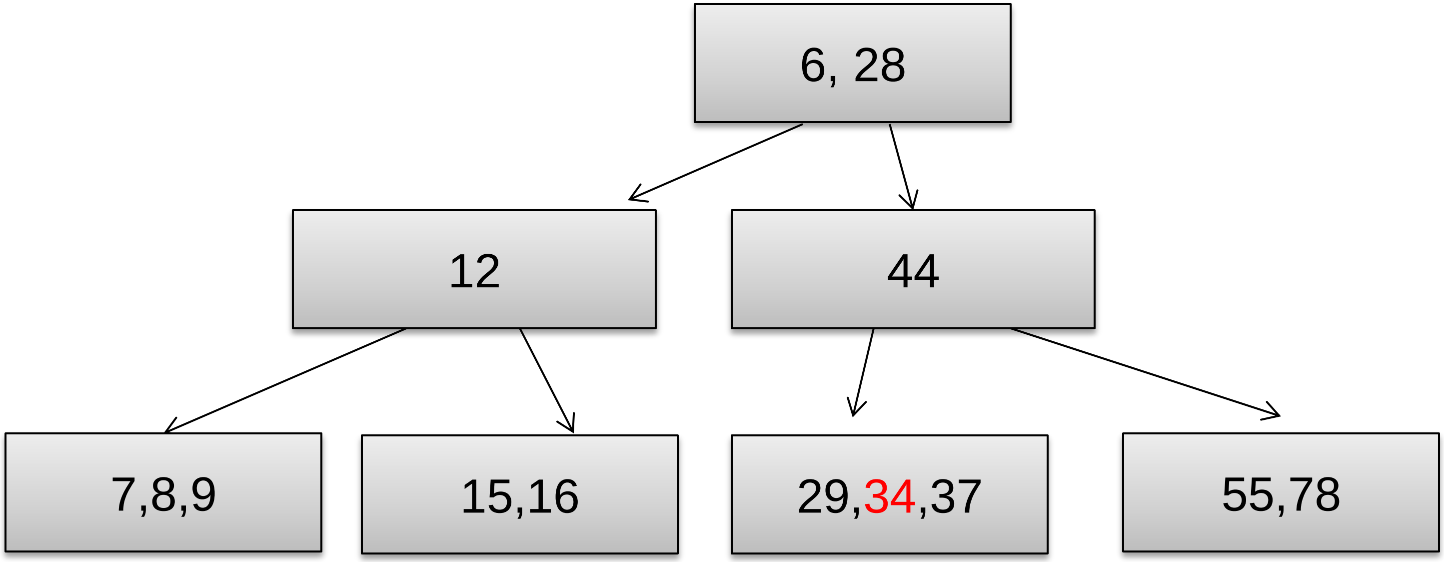
- Einfügen von Schlüssel 30



Big Data

Einfügen mit Knotenaufteilung

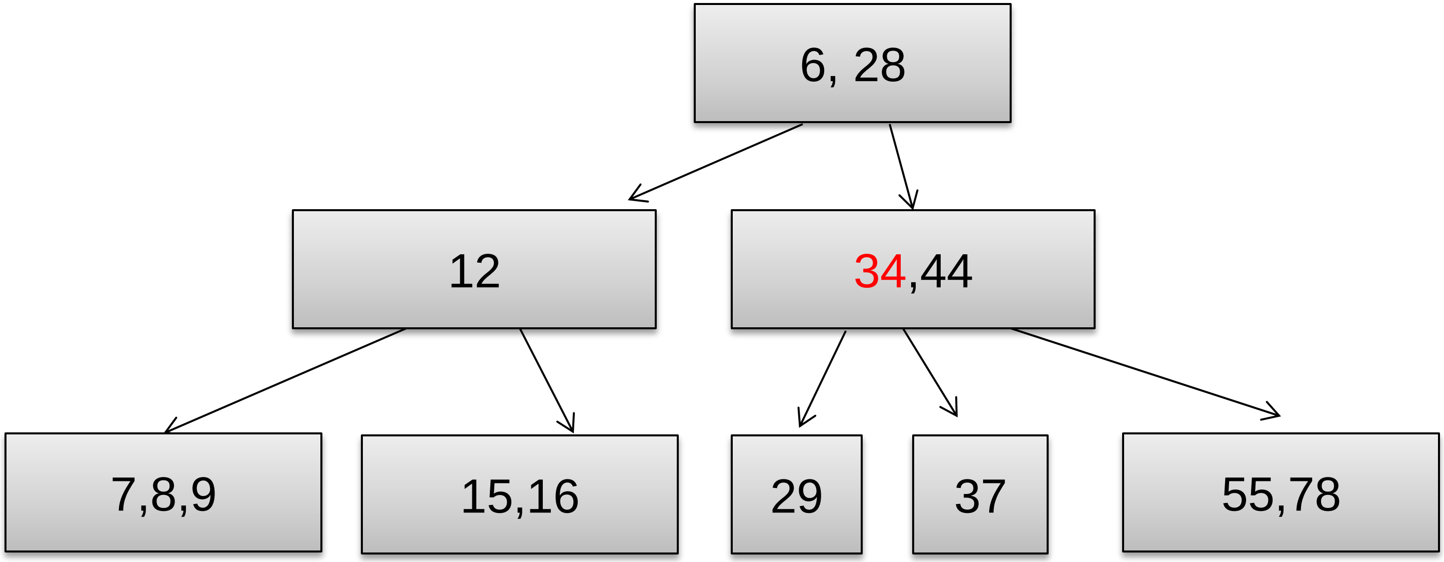
- Einfügen von Schlüssel 30
- Knotensplit



Big Data

Einfügen mit Knotenaufteilung

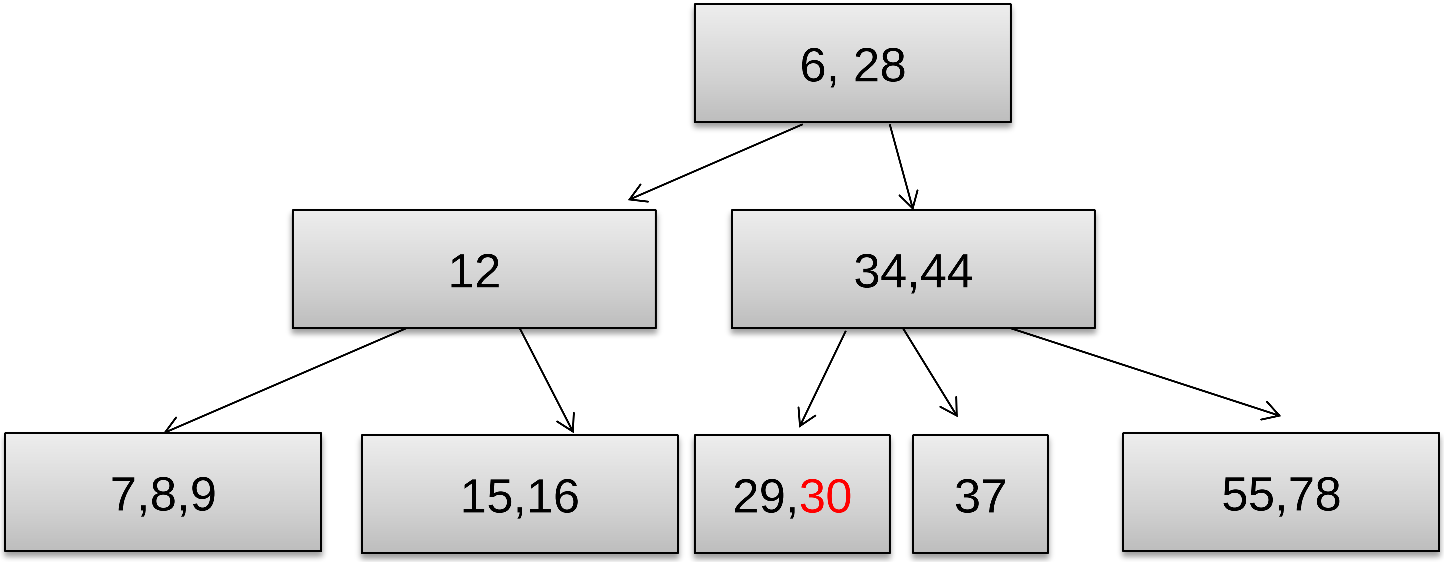
- Einfügen von Schlüssel 30
- Knotensplit



Big Data

Einfügen mit Knotenaufteilung

- Einfügen von Schlüssel 30
- Knotensplit



Big Data

Einfügen(x, k)

➢ Invariante: x ist nicht voll

if x ist ein Blatt **then**
Füge k in die sortierte Schlüsselfolge ein
$n[x] \leftarrow n[x] + 1$
else
$i \leftarrow n[x]$
while $i \geq 1$ and $k < \text{key}_i[x]$ **do** $i \leftarrow i - 1$
$i \leftarrow i + 1$
Disk-Read$(c_i[x])$
if $n[c_i[x]] = 2t - 1$ **then**
Split$(x, i, c_i[x])$
if $k > key_i[x]$ **then** $i \leftarrow i + 1$
Einfügen$(c_i[x], k)$

Big Data

Einfügen(x, k)

- Invariante: x ist nicht voll

if x ist ein Blatt **then**
    Füge k in die sortierte Schlüsselfolge ein
    $n[x] \leftarrow n[x] + 1$
else
    $i \leftarrow n[x]$
    while $i \geq 1$ and $k < \text{key}_i[x]$ **do** $i \leftarrow i - 1$
    $i \leftarrow i + 1$
    Disk-Read$(c_i[x])$
    if $n[c_i[x]] = 2t - 1$ **then**
        Split$(x, i, c_i[x])$
        if $k > key_i[x]$ **then** $i \leftarrow i + 1$
    Einfügen$(c_i[x], k)$

Big Data

➢ Invariante: x ist nicht voll

Einfügen(x, k)

if x ist ein Blatt **then**
  Füge k in die sortierte Schlüsselfolge ein
  $n[x] \leftarrow n[x] + 1$
else
  $i \leftarrow n[x]$
  while $i \geq 1$ and $k < \text{key}_i[x]$ **do** $i \leftarrow i - 1$
  $i \leftarrow i + 1$
  Disk-Read$(c_i[x])$
  if $n[c_i[x]] = 2t - 1$ **then**
    Split$(x, i, c_i[x])$
    if $k > key_i[x]$ **then** $i \leftarrow i + 1$
  Einfügen$(c_i[x], k)$

Big Data

➢ Invariante: x ist nicht voll

Einfügen(x, k)

if x ist ein Blatt **then**
  Füge k in die sortierte Schlüsselfolge ein
  $n[x] \leftarrow n[x] + 1$
else
  $i \leftarrow n[x]$
  while $i \geq 1$ and $k < \text{key}_i[x]$ **do** $i \leftarrow i - 1$
  $i \leftarrow i + 1$
  $\boxed{\text{Disk-Read}(c_i[x])}$
  if $n[c_i[x]] = 2t - 1$ **then**
    $\text{Split}(x, i, c_i[x])$
    if $k > key_i[x]$ **then** $i \leftarrow i + 1$
  $\text{Einfügen}(c_i[x], k)$

Big Data

Einfügen(x, k)

➢ Invariante: x ist nicht voll

if x ist ein Blatt **then**
    Füge k in die sortierte Schlüsselfolge ein
    $n[x] \leftarrow n[x] + 1$
else
    $i \leftarrow n[x]$
    while $i \geq 1$ and $k < \text{key}_i[x]$ **do** $i \leftarrow i - 1$
    $i \leftarrow i + 1$
    Disk-Read$(c_i[x])$
    if $n[c_i[x]] = 2t - 1$ **then**
        Split$(x, i, c_i[x])$
        if $k > key_i[x]$ **then** $i \leftarrow i + 1$
    Einfügen$(c_i[x], k)$

Big Data

Einfügen(x, k)

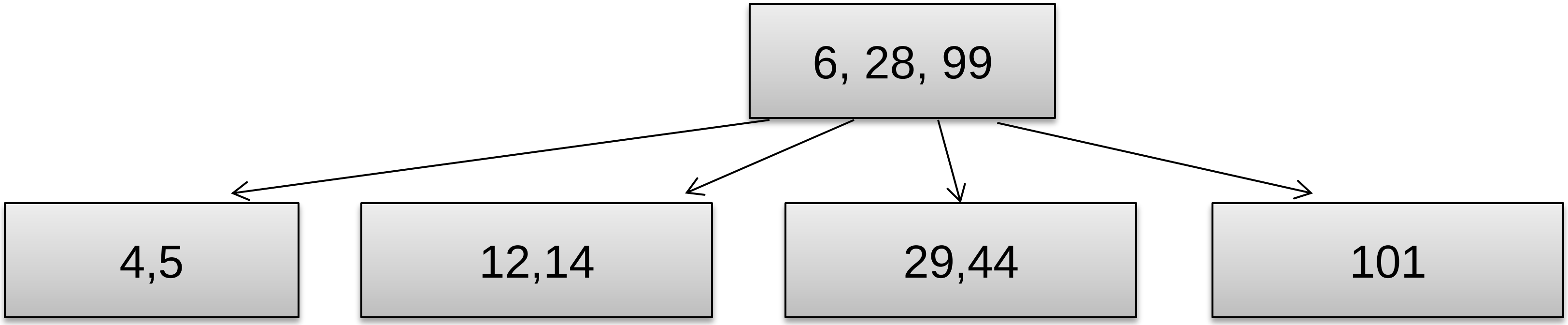
➢ Invariante: x ist nicht voll

if x ist ein Blatt **then**
    Füge k in die sortierte Schlüsselfolge ein
    $n[x] \leftarrow n[x] + 1$
else
    $i \leftarrow n[x]$
    while $i \geq 1$ and $k < \mathrm{key}_i[x]$ **do** $i \leftarrow i - 1$
    $i \leftarrow i + 1$
    Disk-Read$(c_i[x])$
    if $n[c_i[x]] = 2t - 1$ **then**
        Split$(x, i, c_i[x])$
        if $k > key_i[x]$ **then** $i \leftarrow i + 1$
    $\boxed{\text{Einfügen}(c_i[x], k)}$

Big Data

Einfügen mit Wurzelaufteilung

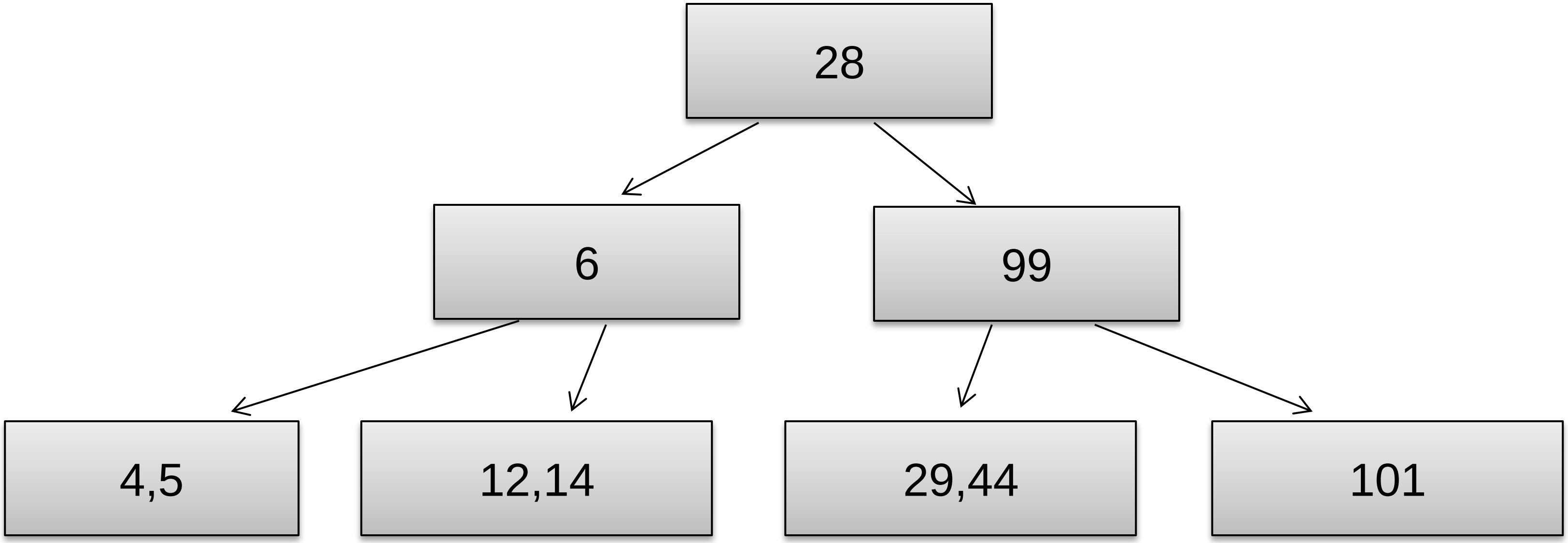
- Einfügen von Schlüssel 30



Big Data

Einfügen mit Wurzelaufteilung

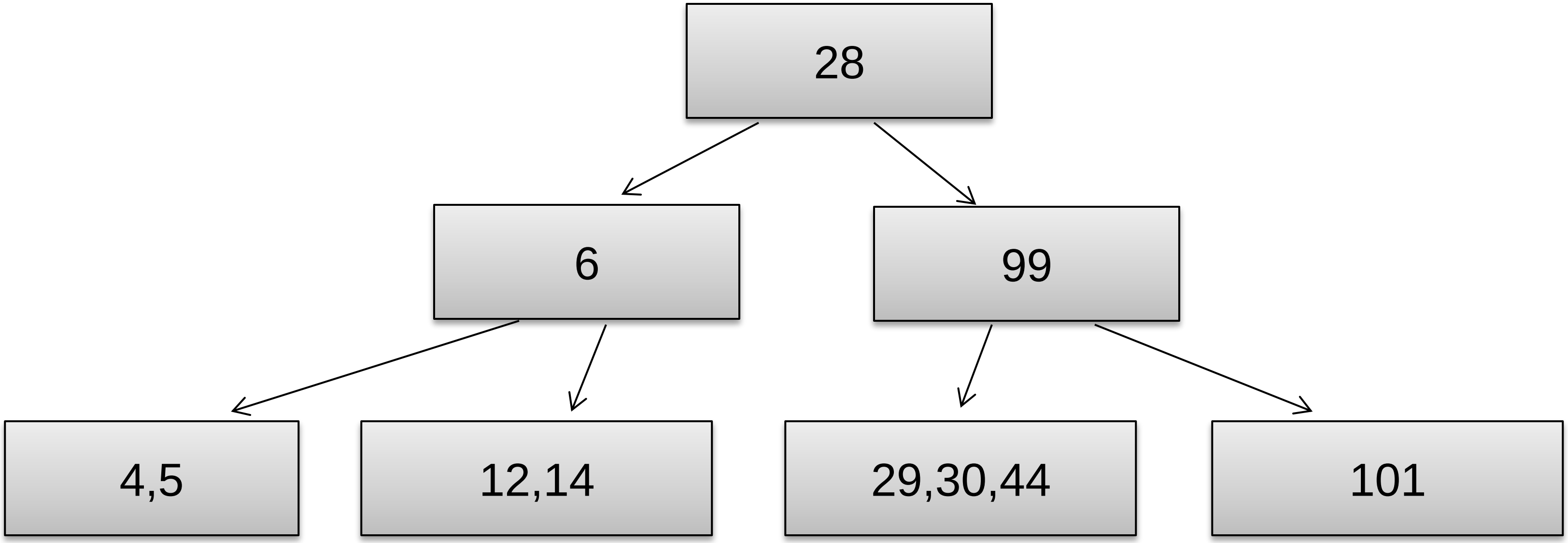
- Einfügen von Schlüssel 30



Big Data

Einfügen mit Wurzelaufteilung

- Einfügen von Schlüssel 30



Big Data

EinfügenVollständig(T, k)

$r \leftarrow \mathrm{root}[T]$

if $n[r] = 2t - 1$ **then**

$s \leftarrow \mathrm{AllocateNode}()$

$\mathrm{root}[T] \leftarrow s$

$n[s] \leftarrow 0$

$c_1[s] \leftarrow r$

$\mathrm{Split}(s, 1, r)$

$\mathrm{Einfügen}(s, k)$

else $\mathrm{Einfügen}(r, k)$

Big Data

$\mathrm{EinfügenVollständig}(T, k)$

$\boxed{r \leftarrow \mathrm{root}[T]}$

$\textbf{if } n[r] = 2t - 1 \textbf{ then}$

$\quad s \leftarrow \mathrm{AllocateNode}()$

$\quad \mathrm{root}[T] \leftarrow s$

$\quad n[s] \leftarrow 0$

$\quad c_1[s] \leftarrow r$

$\quad \mathrm{Split}(s, 1, r)$

$\quad \mathrm{Einfügen}(s, k)$

$\textbf{else } \mathrm{Einfügen}(r, k)$

Big Data

EinfügenVollständig(T, k)

1. $r \leftarrow \text{root}[T]$
2. **if** $n[r] = 2t - 1$ **then**
3. $\quad s \leftarrow \text{AllocateNode}()$
4. $\quad \text{root}[T] \leftarrow s$
5. $\quad n[s] \leftarrow 0$
6. $\quad c_1[s] \leftarrow r$
7. $\quad \text{Split}(s, 1, r)$
8. $\quad \text{Einfügen}(s, k)$
9. **else** $\text{Einfügen}(r, k)$

Big Data

EinfügenVollständig(T, k)

$r \leftarrow \text{root}[T]$

if $n[r] = 2t - 1$ **then**

  $s \leftarrow \text{AllocateNode}()$

  $\text{root}[T] \leftarrow s$

  $n[s] \leftarrow 0$

  $c_1[s] \leftarrow r$

  $\text{Split}(s, 1, r)$

  $\text{Einfügen}(s, k)$

else $\text{Einfügen}(r, k)$

Big Data

$\text{EinfügenVollständig}(T, k)$

$r \leftarrow \text{root}[T]$
$\textbf{if } n[r] = 2t - 1 \textbf{ then}$
    $s \leftarrow \text{AllocateNode}()$
    $\text{root}[T] \leftarrow s$
    $n[s] \leftarrow 0$
    $c_1[s] \leftarrow r$
    $\text{Split}(s, 1, r)$
    $\text{Einfügen}(s, k)$
$\textbf{else } \text{Einfügen}(r, k)$

Laufzeiten (Einfügen)

$\mathbf{O}(th) = \mathbf{O}(t \log_t(n))$ (CPU-)Laufzeit und $\mathbf{O}(\log_t(n))$ Externspeicherzugriffe

Big Data

EinfügenVollständig(T, k)

1. $r \leftarrow \text{root}[T]$
2. **if** $n[r] = 2t - 1$ **then**
3. $\quad s \leftarrow \text{AllocateNode}()$
4. $\quad \text{root}[T] \leftarrow s$
5. $\quad n[s] \leftarrow 0$
6. $\quad c_1[s] \leftarrow r$
7. $\quad \text{Split}(s, 1, r)$
8. $\quad \text{Einfügen}(s, k)$
9. **else** $\text{Einfügen}(r, k)$

Beobachtung

Einfügen wird nur für nicht volle Knoten aufgerufen (Invariante)

Zusammenfassung

Big Data

- Sehr große Datenmengen, die nicht in Hauptspeicher passen
- Externspeicherzugriff erfolgt blockweise
- Laufzeit wird in CPU Laufzeit und Externspeicherzugriffen gemessen

B-Bäume

- Benötigen spezielle Externspeicherdatenstrukturen
- Idee: Knoten sollten einen Block ausnutzen → viele Schlüssel pro Knoten
- Deutlich geringere Anzahl an Externspeicherzugriffen als „einfache“ Binärbäume
- Laufzeit (Suche und Einfügen) $\mathbf{O}(t \log_t(n))$ und $\mathbf{O}(\log_t(n))$ Externspeicherzugriffe
- Löschen ebenfalls effizient möglich